Applications of Genetic Algorithms on theoretical fully-autonomous road systems

University of Birmingham

Sam Barrett, 1803086
sjb786@student.bham.ac.uk
Programme: MSci Computer Science
Supervisor: Miriam Backens
Word count: 10074

April 7, 2021

Todo list

check onenote notebook for general ideas about discussion topics etc.

Abstract

Contents

1 Introduction

1.1 Goals

The overarching goal of this project is to optimally route traffic in the setting of a fully-autonomous road system. This however, is an aspirational goal with many sub-requirements to be fulfilled before it can come to fruition.

Formally this top-level goal can be said to be to answer the question Problem 1.

Top-level Goal

Input:

- A road system $\mathcal{R}$ represented as a graph $G = (V, E)$

  Where each edge $e \in E$ is a section of road defined as the space between two vertices $V^1, V^2 \in V$ which is bounded by two functions $b_1(x)$ and $b_2(x)$ and augmented by a set of obstacles O representing infeasible sections of road space

- a set of agents, $A = \{a_1, \ldots, a_n\}$

- a set of vertex pairs representing start and goal points for each agent:

  $\mathcal{V} = \left\{ (V_1^1, V_1^2), \ldots, (V_n^1, V_n^2) \right\}, V_{i \in [1 \ldots n]}^{j \in [1,2]} \in V$

  where the route for agent a_i is from vertex V_i^1 to V_i^2

Question: What is the set of optimal routes $\mathcal{V}_{optimal}$, where the ith element is defined as a set of linked Bézier curves connecting V_i^1 and V_i^2 through feasible space?

Problem 1

We can then split this into more manageable sub-goals. The sub-problem of generating a route through a section of road is defined in Problem 2. This

can be further decomposed into the selfish routing of a single agent through a section of road defined in Problem 3

Sub-Goal: Cooperative route planning

Input:

- A start point P_{start} and a goal point P_{goal}

- A section of road as defined in Problem 1

- A knowledge of all routes being planned to be executed concurrently.

Question: What is the optimal route, in the form of a Bézier curve, between these two points s.t. it does not collide with any other agents or pass through any infeasible regions in the road space?

Problem 2

Sub-Goal: Single agent route planning

Input:

- A start point P_{start} and a goal point P_{goal}

- A section of road as defined in Problem 1

Question: What is the optimal route, in the form of a Bézier curve, between these two points s.t. it does not pass through any infeasible regions in the road space?

Problem 3

1.1.1 Requirements

The goal of this project was not to produce a production-ready system, but instead, to investigate the plausibility of GAs on the real future possibility of completely autonomous road networks. As such I feel it useful to outline the theoretical requirements of a production grade system. To do this formally

Sub-Goal: Bézier curve generation

Input:

- A start point P_{start} and a goal point P_{goal}

- A section of road as defined in Problem 1

Question: What is the optimal route, in the form of a Bézier curve, between these two points s.t. it does not pass through any infeasible space?

Problem 4

I will employ Propositional and Temporal logic.

1. The system should never return a set of routes such that, for any time $t \in T, \forall i \in n, \forall j \in n, j \neq n, I_i(t) = I_j(t)$. I.e. for any time, no routes should inhabit the same point, meaning there are no collisions in the planned routes.

This is wrong, correct this or remove it outright

2 Background

2.1 Genetic Algorithms

2.1.1 History

Genetic algorithms are a type of meta-heuristic optimisation technique that employ the same rationale as classical Evolution seen in nature.

Genetic Algorithms can trace their origins back to the late 1960s when they were first proposed by John Holland, though he then referred to them as *Genetic Plans*[1]. Holland went on to write the first book on the subject titled *Adaptation in Natural and Artificial Systems*[4] in 1975. The field did not find much reception with Holland stating in the preface to the 1992 rerun:

> *"When this book was originally published I was very optimistic, envisioning extensive reviews and a kind of 'best seller' in the realm of monographs. Alas! That did not happen."*

However, in the early nineties, Genetic algorithms surged in popularity along with the area of Artificially Intelligent planning as a whole, leading to Holland republishing his book and solidifying his position as the field's founder.

2.1.2 Definition

In a general sense, optimisation techniques work to find the set of parameters $\mathcal{P}$ that minimise an objective function $\mathcal{F}$. Genetic algorithms approach this by representing these sets as individuals in a population, P. Over the course of multiple generations, the best solutions are determined and promoted until termination criteria are met or the maximum number of generations is reached.

As our candidates are essentially a collection of parameters to the function we are trying to optimise, we can extend our metaphor further by mapping each element of a individual to a *gene* in a individual's genome.

The representation we use in a GA is problem specific. Often we have to provide functions to facilitate the mapping between the problem specific set of possible solutions and the encoded genotype space in which we optimise. The most basic representation being a string of binary numbers.

An individual's characteristics and genetic information is normally encapsulated within the Phenotype. Here not only the genetic information of

---

[1]This distinction was made to emphasise the *"centrality of computation in defining and implementing the plans"*[4]

the Genotype but also additional information, such as fitness, is stored in order to prevent it from being re-calculated as often.

Genetic algorithms are both *probabilistically optimal* and *probabilistically complete*[1] meaning that: given infinite time, not only will the algorithm find *a* solution, (if one exists), it will find **the** optimal solution from the set of all possible solutions, $\mathcal{P}^*$.

Algorithm 1: Modern Generic Genetic Algorithm

Result: Best Solution, p_{best}

Generate initial population, P_0 of size n;

Evaluate fitness of each individual in P_0, $\{F(p_{0,1}, \ldots, p_{0,n})\}$;

while *termination criteria are not met* **do**

    Selection: Select individuals from P_t based on their fitness;

    Variation: Apply variation operators to parents from P_t to produce offspring;

    Evaluation: Evaluate the fitness of the newly bred individuals;

    Reproduction: Generate a new population P_{t+1} using individuals from P_t as well as the newly bred candidates.;

    $t{+}{+}$

end

return p_{best}

As you can see from Figure 2.1 and Algorithm 1 the overall shape of GAs has not changed substantially over the course of the past 50 years. Being comprised of a series of operations that a starting population is piped through until criteria are met.

2.1.3 Genetic Operators

In the following section I will outline the various genetic operations that take place in a GA, they can be seen in Algorithm 1. Any operators used and analysed in Chapter 4 will have more detailed explanations of their process.

> Expand definitions of operators used in implementation section

Selection

The selection procedure is the process by which the next generation of individuals is created from the current population. Individuals are selected relative to their fitness as determined by the Objective (fitness) function $\mathcal{F}$.

Some methods select only the best solutions by fitness. Others employ a more stochastic approach, such as roulette wheel selection, to increase diversity and reduce complexity.

Fitness Proportional Selection   Fitness proportional or Roulette wheel selection is a popular selection operator. It uses fitness to assign selection probability to each individual in a population.

1 Set $t = 0$ and initialize $\mathcal{B}$ by selecting M structures at random from $\mathcal{C}_1$ to form $\mathcal{B}(0) = (A_1(0), \ldots, A_M(0))$.

2.1 Observe and store the performances $(\mu_1(0), \ldots, \mu_M(0))$ to form $\mathcal{V}(0)$.

2.2 Calculate $\hat{\mu}(0) = \sum_{h=1}^{M} \mu_h(0)/M$.

2.3 Observe the performance $\mu_E(A'(t))$.

2.4 Update $\hat{\mu}(t)$ by calculating $\hat{\mu}(t) + \mu_E(A'(t))/M - \mu_{i(t)}(t)/M$.

2.5 Update $\mathcal{V}(t)$ by replacing $\mu_{i(t)}(t)$ with $\mu_E(A'(t))$.

3 Increment t by 1.

4 Define the random variable $Rand_t$ on $\mathcal{I}_M = \{1, \ldots, M\}$ by assigning the probability $\mu_h(t)/\hat{\mu}(t)$ to $h \in \mathcal{I}_M$. Make one trial of $Rand_t$ and designate the outcome $i(t)$.

5.1 Apply simple crossover (as defined in section 6.2 and extended in section 6.3) to $A_{i(t)}(t)$ and $A_{i'(t)}(t)$ with probability P_C, where $A_{i'(t)}(t)$ is determined by a second trial of $Rand_t$. Select one of the two resultants at random (equilikely) and designate it $^1A(t)$ (where the order of attributes in the resultant is that of $A_{i(t)}(t)$).

5.2 Apply simple inversion (as defined in section 6.3) with probability P_I, yielding $^2A(t)$.

5.3 Apply mutation (as defined in section 6.3) to $^2A(t)$ with probability $c_t \cdot {}^1P_M$, yielding $A'(t)$.

6.1 Assign probability $1/M$ to each $h \in \mathcal{I}_M$ and make a random trial accordingly; designate the outcome $j(t)$.

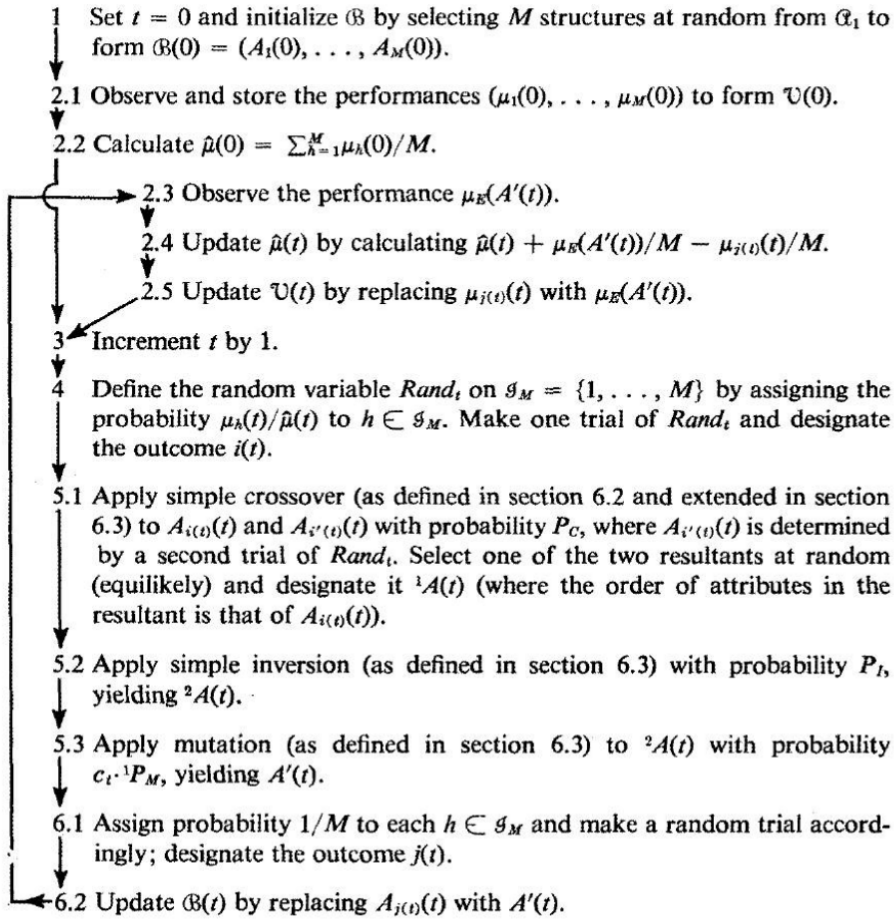**6.2** Update $\mathcal{B}(t)$ by replacing $A_{j(t)}(t)$ with $A'(t)$.

Figure 2.1: GA outlined in Holland's Original Book[4]

10

The probability of selection for an individual i with fitness $\mathcal{F}(i)$ can be expressed mathematically as:

$$p_i = \frac{\mathcal{F}(i)}{\sum_{j=1}^{N} \mathcal{F}(j)} \tag{2.1}$$

Where N is the size of the population P

This is a simple approach but performs well and has very little performance overhead.

Ranked Selection   Ranked selection is an alternative selection method, it works on the assumption that the individuals in your population are closely grouped in fitness. The main difference of this method when compared to fitness proportional selection is that it ranks individuals based on relative fitness rather than absolute fitness.

The procedure can be written mathematically as:

Given a population P of size n ordered by fitness. We select the top γ-ranked individual, x_γ with a probability $p(\gamma)$. Where γ is the rank and $p(\gamma)$ is the ranking function.

The ranking function can take different forms including both linear and exponential ranking.

extend list to include all functions used

Linear ranking is defined as:

$$p(\gamma) = \frac{\alpha + (\beta - \alpha) \cdot \frac{\gamma}{n-1}}{n} \tag{2.2}$$

Where:

- $\sum_{\gamma=0}^{n} p(\gamma) = 1$

- $\alpha + \beta = 2$

- $1 \leq \beta \leq 2$

Variation

Variation in a GA is the process of altering the genome individuals to further explore the search space via stochastic local search. We perform this using two distinct sub-operations: Mutation and Crossover. Here we can view crossover as the *breeding* process and mutation as resembling the natural tendency for DNA to mutate over the course of generations.

Mutation In mutation we alter each gene with a set probability p_m known as the *mutation rate*. A standard value for a mutation rate is $\frac{1}{L}$ but it can fall anywhere in the range $p_m \in [\frac{1}{L}, \frac{1}{2}]$ where L is the length of the genome.

A low value for p_m a new individual which can be shown to be *close* to it's parents in the search space relative to their *Hamming distance*[2] if using a Binary coded GA. In real-coded GAs they can be shown to be close by the Euclidean distance between them.

In binary coded GAs we alter a given gene by *flipping* it's value. In real-coded GAs mutation operators include:

- Uniform Mutation

- Non-Uniform Mutation

- Gaussian Mutation

Uniform Mutation In uniform mutation, we select a parent p at random and replace a randomly selected gene $c_i \in p$ with a uniformly random number $c_i' \in [u_i, v_i]$ where u_i and v_i are set bounds.

Gaussian Mutation In Gaussian mutation, we select a parent p at random and randomly select a gene $c_i \in p$. We replace c_i with a new value c_i' which is calculated as follows:

$$c_i' = \min(\max(\mathcal{N}(c_i, \sigma_i), u_i), v_i) \qquad (2.3)$$

Where $\mathcal{N}(c_i, \sigma_i)$ is a Gaussian distribution with a standard deviation σ_i and a mean of c_i. σ_i may depend on the length of the interval bound, $l_i = v_i - u_i$, typically (and in my implementation) $\sigma_i = \frac{1}{10} l_i$

Crossover

Crossover is a binary operation, taking two randomly selected *parents* from the population P_t with a probability $p_c \in [0, 1]$

There are two major forms of crossover for binary coded GAs: $n > 1$ point crossover and Uniform crossover.

For real-coded GAs you instead have a selection of Crossover operators including:

- Flat crossover

- Simple crossover

- Whole arithmetic crossover

---

[2]Hamming Distance: A metric for comparing two binary data strings. The Hamming distance between two strings is the number of bit position in which they differ.

- Local arithmetic crossover

- Single arithmetic crossover

- BLX-α crossover

Simple (one point) Crossover For simple crossover, randomly select a crossover point, $i \in \{1, \ldots, n\}$. All values before this point are swapped between the two parents. For 2 parents, $p_1 = \left\{ x_1^{[1]}, \ldots, x_n^{[1]} \right\}$ and $p_2 = \left\{ x_1^{[2]}, \ldots, x_n^{[2]} \right\}$ this can be represented as:

$$p_1' = \left\{ x_1^{[1]}, x_2^{[1]}, \ldots, x_i^{[1]}, x_{i+1}^{[2]}, \ldots, x_n^{[2]} \right\} \tag{2.4}$$

$$p_2' = \left\{ x_1^{[2]}, x_2^{[2]}, \ldots, x_i^{[2]}, x_{i+1}^{[1]}, \ldots, x_n^{[1]} \right\} \tag{2.5}$$

Evaluation

After developing potential new individuals, the fitness of these new individuals is calculated using the objective function, $\mathcal{F}$

Reproduction

Here the next generation of individuals is constructed. There are multiple potential methods employed here with the simplest being to just replace the lest fit individuals with additional copies of the fitter individuals, be they pre-existing or newly generated.

In this stage various heuristics or alternative strategies can be implemented to speed up or slow down the convergence rate. Whilst their fitness may be low, having a diverse population allows for more of the search space to be explored. If the algorithm converges too quickly it may get *stuck* in local minima.

2.2 Bézier Curves

In my implementation, I utilise Bézier curves to encode complex route arcs between a series of points. Here I will briefly outline their construction and mathematical basis.

Bézier curves were popularised by and named after French auto-body[3] designer Pierre Bézier in the 1960s and are commonly found in computer graphics today. They are parametric curves made up of a series of *control points* which contort the shape of a line to produce a curve of nearly any

---

[3]so I feel it is quite fitting that they find use in 21st century automotive problems

shape. Although their mathematical basis was established in 1912 by Sergei Bernstein in the form of Bernstein polynomials[5].

A Bézier curve is said to have a degree of $n - 1$ where n is the number of control points including the start and end point of the curve.

2.2.1 Formal Definition

Bézier curves can be simply explicitly defined for degrees of 1 through 4, however, the general case of n points is more useful to us. In their general form they can be defined recursively or explicitly. I implement the recursive version as it is much more readable for a programming language that supports recursion.

They are defined recursively as follows:

$$\mathbf{B}_{P_0}(t) = P_0 \tag{2.6}$$

$$\mathbf{B}_{P_0, P_1, \ldots, P_n}(t) = (1 - t)\mathbf{B}_{P_0, P_1, \ldots, P_{n-1}}(t) + t\mathbf{B}_{P_1, P_2, \ldots, P_n}(t) \tag{2.7}$$

Where the parameter t is in the range of $[0, 1]$. Therefore, the smoothness of the curve is determined by the granularity of the parameter when realising the curve.

2.2.2 Length of a Bézier curve

The length of a Bézier curve is a non-trivial calculation and is best approximated as:

$$L = \frac{(2L_c + (n - 1)L_p)}{n + 1} \tag{2.8}$$

Where L_c is chord length and L_p is polygon length. This was proven by Jens Gravesen[6] in 1997.

Chord length is given by:

$$L_c = \sqrt{\left(P_{0_x} - P_{n_x}\right)^2 + \left(P_{0_y} - P_{n_y}\right)^2} \tag{2.9}$$

and polygon length by:

$$L_p = \sum_{i=0}^{n-1} \sqrt{\left(P_{i_x} - P_{{i+1}_x}\right)^2 + \left(P_{i_y} - P_{i_y}\right)^2} \tag{2.10}$$

Or, alternatively:

$$L_p = \sum_{i=0}^{n-1} L_c(B_{P_i, P_{i+1}}) \tag{2.11}$$

2.2.3   Subdivision of n-degree Bézier curves

Subdivision of Bézier curves is a key stage in the process of finding any intersections between two Bézier curves. The process of subdivision is performed using De Casteljau's algorithm. This algorithm was developed by Paul de Casteljau while working at Citroen in 1959 to work with the family of curves that would later be formalised into Bézier curves by Pierre Bézier[4].

It is a simple procedure, given a Bézier curve B, it can be split at any arbitrary point $t \in [0, 1]$ into two curves with control points of the form:

$$\beta_1 = B_{P_0}(t), B_{P_0,P_1}(t), \ldots, B_{P_0,\ldots,P_n}(t) \tag{2.12}$$

$$\beta_2 = B_{P_n}(1-t), B_{P_n,P_{n-1}}(1-t), \ldots, B_{P_n,\ldots,P_0}(1-t) \tag{2.13}$$

> Should I change this superscript notation? subscript is being used to denote which control points are being considered.

2.3   Fully Autonomous Road Networks

Fully autonomous road networks are by no means a novel concept. They have appeared in fiction since the mid 20th century and have been a real possibility for the past decade. In a fully autonomous road networks *(FARN)*s, all vehicles are self-operating with their routing either being determined on a vehicle-by-vehicle *selfish* basis or by a larger system managing routes for all nearby agents.

In the latter system protocols that promote the net increase in efficiency can be implemented. However, this sort of system will require a form of government mandate as a universal routing protocol would need to be established and implemented by all auto manufacturers; this is no small undertaking.

2.4   Miscellaneous

2.4.1   Bounded Boxes

A bounded box of a curve is defined as the box constructed from the points $(x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{min}), (x_{max}, y_{max})$ Where $x_{min,max}$ and $y_{min,max}$ are the minimum and maximum x and y values from all control points respectively.

---

[4]Bézier curves have also been called de Casteljau curves as much of his work preceded that of Pierre Bézier's
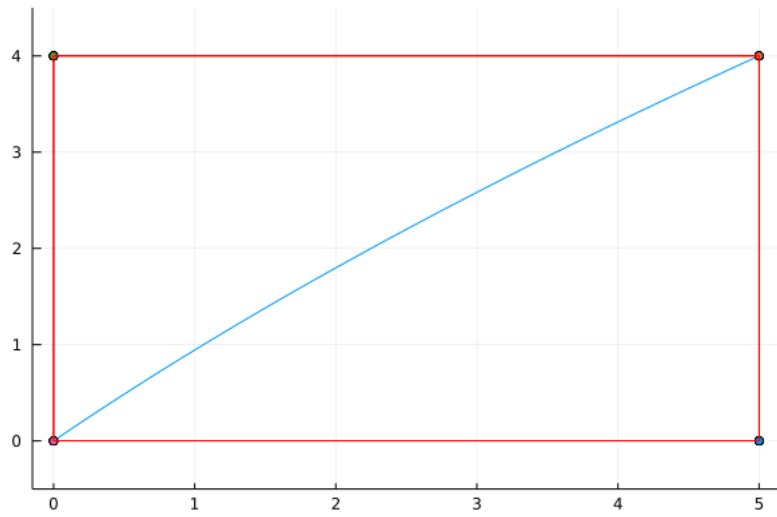
Figure 2.2: A bounding box for a Bézier curve of degree 2

3 Literature Review

3.1 Genetic algorithms for route generation

Classical Genetic algorithms have seen research and applications in a number of fields over the past 40 years ranging from Heart disease diagnosis[7] to predicting the strength of concrete under various conditions[8].

There is a substantial and growing body of research specifically looking at the applications of GAs in route planning for autonomous agents, however, many are focusing on either abstract robotic agent planning in discrete search spaces or failing to take into account the wider environment of multiple agents.

3.1.1 Genetic algorithms for cooperative route generation

Cai & Peng - Cooperative Coevolutionary Adaptive Genetic Algorithm in Path Planning of Cooperative Multi-Mobile Robot Systems[2]
In this 2001 paper, Cai & Peng give an example of a Cooperative Coevolutionary Adaptive Genetic Algorithm (CCAGA) for planning routes for multiple autonomous agents.

Their approach features a discrete grid-based search space featuring rectangular obstacles which the routes must avoid. Their chromosomes are encoded using real-values representing the x and y coordinates of the search grid.

Their fitness function has two stages, in the first generation the fitness of a candidate solution is purely determined by the global *optimality* of the route. In all subsequent generations the fitness also takes into account the best routes from each of the other sub-populations.

They conclude that their approach yields good robustness and convergence as well as being suitable for parallel execution, allowing for systems employing this GA approach to run very efficiently.

Raul Kala - Optimization Based Planning 2016[9]   In his book, Kala proposes a GA as an embedded component of a larger planning system which also relies on Djiksra's algorithm for macro-level route planning along with the classical autonomous vehicle real-time collision avoidance toolbox (radarr, sonarr, etc.). In order to achieve a system capable of planning for multiple agents concurrently, Kala proposed the use of these real-time collision avoidance tools along with embedding *traffic rules* as heuristics into the Djiksta-based component. These traffic rules included "driving on the left" and "overtaking on the right" etc. This proposed system, while similar

make reference to this method being altered for use in my solution(in approach section?)

to mine, differs in scope. I propose a system for planning routes for a set of autonomous agents within a fully-autonomous road system, meaning agents will not (in theory) interact or encounter any other vehicles which are not a part of the planning network.

Cruz-Piris et al. - Automated Optimsization of Intersections Using a Genetic Algorithm[10] A paper published by Cruz-Piris et al. investigates the applications of (binary coded) Genetic Algorithms on routing traffic through busy intersections. Their proposed system is shown to have improvements in traffic throughput of anywhere from 9% to 36%.

Their approach involved several assumptions, many of which my research shares. Namely, they assumed vehicles do not stop at any point and that their speed remains constant at all points in time.

Their approach to modelling the solution space takes advantage of the fairly uniform and regular structure of large intersections, and although they work on modelling irregularly shaped intersections, this TCA model begins to fall apart when the size of each cell cannot contain a vehicle and requisite surrounding space or if input/output lanes do not lie in a regular grid pattern. Facilitating irregularly shaped intersections seems to be a manual process, not a particularly viable method when you consider the number of unique intersections across the world or even just the Continental United States!

This approach seems to work relatively well for throughput optimisation problems such as junctions but does not scale well for less structured scenarios where all origins and goals cannot be known beforehand and are not necessarily the same for all subsequent runs.

Roberge et al. - Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU[3] In this 2018 paper an interesting proposition was made, greatly increasing efficiency and viability of real-time planning via GAs through massively parallelising its core processes to run on GPUs. Their implementation shows a 290x times speedup compared with conventional sequential CPU execution. GAs lend themselves quite nicely to parallel execution as we are applying the same operators to multiple individuals of the same shape.

3.2 Bézier curves for route generation

Chen et al. - Quartic Bézier Curve based Trajectory Generation for Autonomous Vehicles with Curvature and Velocity Constraints In this 2014 paper, Chen et. al research the efficacy of quartic (4-degree) Bézier curves for planning continuous routes between points for autonomous vehicles. In order to find an optimal solution they employ sequential quadratic programming.

18

In their approach Chen et al. also take into account the orientation, change in orientation and velocity of the routes generated in order to assure all generated routes are *safe* for occupied vehicles to follow.

This approach appears to be extremely computationally efficient on the toy examples provided in the paper, on relatively low performance hardware 560 iterations can be computed in around 500ms!

There is no mention, however, to a cooperative element. Explicitly solving quadratic programming tasks is known to be **NP**-complete as shown by Vavasis in 1990[11] and adding onto that the requirement that all solutions interact nicely will only increase the real-world runtime of any solution. A real-world scenario with hundreds of vehicles may suffer from an incredibly high runtime, hardware requirement or both.

is this even correct to say?

3.3 Fully Autonomous Road Networks

*FARN*s have the possibility of improving travel in a number of different areas.

They have the potential of greatly reducing travel times by efficiently routing all vehicles with the goal of reducing the net travel time. Such a system would also be able to respond much faster than human drivers, allowing for large increases in permissible vehicle velocities thus, further increasing efficiency.

By extension of their higher efficiency, *FARN*s will also have a marked effect on vehicular energy consumption. The potential effects are summarised nicely by Vahidi and Sciarretta[12] where they show the use of vehicle automation could cause anywhere from a halving of "energy use and greenhouse gas emission" in an "optimistic scenario" to doubling them depending on the effects that present themselves. The increase in highway speed whilst increasing travel efficiency is predicted by Brown et al.[13] and Wadud et al.[14] as increasing energy use by anywhere from 5% to 30%

4 Approach

4.1 Single Agent Approach

The goal of this section was to design a system that fulfilled the requirements outlined in Problem 3

The general shape of a GA as seen in Algorithm 1 is the same for almost all problems. But each individual operator implementation is domain specific.

4.1.1 Individual Encoding

As such, I first had to implement a method to generate an initial population. In order to do this, I had to define my programmatic representations of the phenotypes and genotypes of my individuals. I decided to base my genotypic structure on the that described by Kala[1].

In this approach, the genotype is abstractly represented as a Bézier curve (Section 2.2). This is more concretely translated into a real-valued string, interleaving the x and y coordinates of each control point. Taking the form:

$$\boxed{P_{0_x}} \ \boxed{P_{0_y}} \ \boxed{P_{1_x}} \ \boxed{P_{1_y}} \ \boxed{\cdots} \ \boxed{P_{n_x}} \ \boxed{P_{n_y}}$$

Figure 4.1: Individual genotype as real-valued string

In the above form the descriptions of genetic operators found in Section 2.1.3 should be sufficiently detailed to re-implement.

Initially my Phenotype contained no additional information. However, I still created the distinction to allow for additional properties to be encapsulated during development.

Phrasing? Correct? should I elaborate?

4.1.2 Population Generation

Population generation theoretically can be very simple and generate very little diversity, relying on the other operators to explore the search space. However, the more diverse a set of initial routes one can generate, the quicker the system will converge to an optimal solution.

As such we want to ensure that both the number of control points as well as their position are properly varied.

The first and last control points **must** be the starting and goal position for the agent. In a Bézier curve these are the only two points that the curve definitely passes through. We must also ensure that the x coordinates of

the control points are in order, so as to avoid routes which *double back* on themselves. This can also occur as the result of other operators such as crossover and mutation, we can fix this using a simple repair operator which re-orders points based on their x positions.

4.1.3 Fitness assessment

The foundation of my Fitness function is the length of each route, i.e. the length of a given n degree Bézier curve. This calculation can be accurately approximated using Equation 2.9 as the true calculation is expensive and does not generalise nicely to n degrees[6].

Minimising the length of a route is a good way of ensuring generated routes are direct, however, other values and heuristics are required to maintain other requirements such as to make sure changes in direction are not too severe or that a route does not pass through infeasible space (avoiding obstacles etc.).

I initially experimented with an alternative coordinate space which enabled me to completely enclose the search space within the area of the road, eliminating the possibility of generating routes that left the *road*. This was inspired by the "Road coordinate axis system" described by Kala in [9]. However I abandoned this concept as it did not appear to increase performance and introduced overhead when visualising or generating the routes.

Instead I defined each section of road using the following Structure:

```julia
struct Road
    boundary_1::Function
    boundary_2::Function
    obstacles::Array{Obstacle}
    length::Real
end
```

This approach allows me to define the upper and lower boundary of a road as any function, straight roads can be defined as a pair of straight lines a certain width apart, and arbitrarily complex roads can be represented using arbitrarily complicated functions.

For example the road seen in Figure 4.3 had boundary functions:

$$b_1(x) = 0$$
$$b_2(x) = 5$$

And the curved road section seen in Figure 4.2 functions:

$$b_1(x) = 2\cosh(0.1x) - 2$$
$$b_2(x) = 2\cosh(0.12x) + 8$$

Where in both cases the length of the road (and therefore upper bound of x) was 25, the lower bound of x was 0.
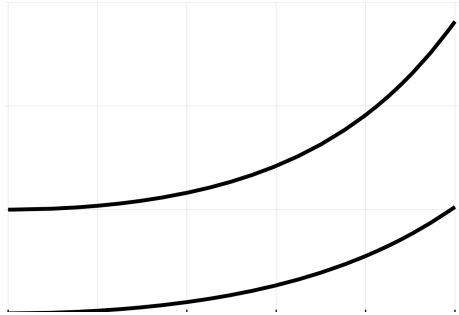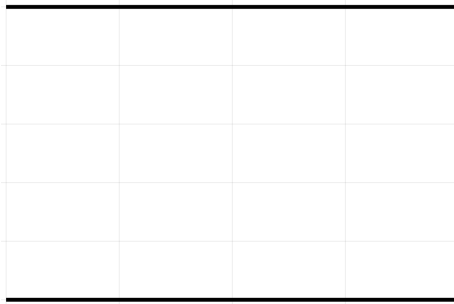

Figure 4.2: A curved road

Figure 4.3: A straight road

Road Obstacles   I have implemented two basic `Obstacle` types (`Rectangle` and `Circle`) allowing me to create regions of infeasible space within the valid road-space.

Infeasible space is then calculated by taking a sample of 500 points along a given curve B and checking which (if any) lie within any of the obstacles.

A Bézier curve is then constructed using two applications of De Casteljau's algorithm (See Section 2.2.3) using values for t derived from the subsample points. An illustration of this process can be seen in Figure 4.4. The length of this curve is weighted and added to the overall fitness of the candidate solution.

A similar process is performed to determine how much of a curve passes too closely to an infeasible region, this is penalised with a separate, lower, weighting.

Once the initial population is generated, the main algorithmic loop can begin, this is the repeated application of the selection, crossover, mutation and evaluation operators (See Section 2.1.3) until set termination criteria

replace with plotted/ computer designed version

22

Figure 4.4: Illustration of finding infeasible section of curve

are met or the maximum number of generations is completed, seen in Algorithm 1.

The final set of routes is finally filtered by the predicate `infeasible-distance`$(i) = 0$

4.2 Multi-agent Approach

So far we have only concerned ourselves with planning a route through a road-space for a single agent. However, in the real world, roads are seldom occupied by a single vehicle and as such we must consider how to efficiently plan a set of routes for a set of agents between between a set of coordinate pairs, such that, our agents do not collide at any point in time.

There are different approaches we can take to this problem.

refer to lit rev sections discussing approaches by Kala[1] and Cai & Peng [2]

My initial approach to a collaborative planning system was somewhat inefficient. The system operated by planning routes for each agent sequentially, using a growing *context* to keep track of the routes that had already been finalised. The first agent to be planned was not concerned with avoiding collisions, as, as far as it was concerned, there were no other agents in the system. Each subsequent agent checked if any of its candidate solutions intersected with and of the routes planned before, a very high penalty was applied to any routes with collisions.

23

This system was very inefficient with the final routes requiring a large number of checks to be carried out by my `bezInt` function, which itself suffered from

exponential time (and space) complexity.

This system was improved upon by splitting the process over multiple threads, each agent being given a separate thread, communicating their most up-to-date plans via a shared array, S. Agent i will store its current fittest route in $S[i]$, the fitness of any candidate checks for collisions with each route in S. This system may conduct more checks but it removes the problem of prioritising the first route to the planned and reduces the overall runtime of the system approximately proportionally to the number of threads available.

4.2.1 Collision Detection

Collision detection is one of the core obstacles to a viable cooperative route planner. You must be able to certify that your resulting set of routes do not collide at any point in time, else the entire system fails.

Simply detecting intersections is not enough as two routes can intersect but only collide if they pass through the same point at the same time. We therefore require some notion of time when realising our Bézier curve routes.

I made the assumption that each agent in my system travels at a uniform constant speed. To remove this assumption we would need to include a velocity-time profile in the genotype of each individual, this is an area for further research.

With this assumption, we can now say that two routes collide if they intersect and the distance from their respective origins and the point of intersection is the same. We have now reduced this problem to finding a point of intersection between two Bézier curves, this is still a non-trivial task.

Bézier Curve intersection

The commonly employed technique for finding intersections between two n-degree Bézier curves is repeated recursive subdivision. Whilst it is possible to precisely calculate the point of intersection it is very difficult and computationally expensive and does not generalise well to n degrees. For example, to numerically find the intersection two curves of degree 3 we must solve a 9th order equation, producing unstable results. This is described by Bézout's theorem which states that two planar algebraic curves of degrees d_1 and d_2 will have up to $d_1 d_2$ intersection points.

In their 2006 paper Yap[15] proposed an efficient method for finding the intersection of two Bézier curves through a 2 stage process surrounding repeated subdivision.

During development I attempted to use a Bézier curve library (`libbezier`)[16] written for python but featuring a core written in Fortran, providing a C ABI. Julia, featuring interoperability with C and Fortran, allowed me to make direct calls to the Fortran subroutines and C functions. Ultimately, however, I encountered too many issues stemming from this library's inability to be run in parallel along with seemingly random segfaults which I struggled to diagnose through two levels of language abstraction (Fortran $\rightarrow$ C $\rightarrow$ Julia); this led me to write my own functions natively in Julia, although replicating the same accuracy as the nearly 4000 lines of Fortran proved challenging.

In order to speedup my Bézier Intersection function I implemented a number of time saving measures:

1. If the convex hulls of two curves B_1 and B_2 intersect, i.e. some amount of the area of the convex hull of B_1 ($CH(B_1)$), lies inside the convex hull of B_2 ($CH(B_2)$) then there may be some intersection and as such, further checks are required. If however, this is not the case then (likely) the two curves do not intersect and the function can exit early.

   This is a simplified version of the logic presented by Yap in his *Micro Phase* of intersection detection. However, in order to cover all edge cases and make this a reliable rule to follow a lot more precomputation is required (See Elementary curves & curve coupling in [15]).

   As such, I ended up removing this feature as the additional work required was too high for this project.

2. I instead instituted a simpler heuristic using bounded boxes (See Section 2.4.1). Using the `Luxor` library I was able to detect whether two bounded boxes intersected, if they did not, further checks can be skipped.

3. If two curve segments have already been checked for intersection, why bother checking them again?

   This question was solved by trading computation time for memory, by constructing a hash table linking curve pairs to a tuple of intersection curve segments (if present) and a boolean, when checking two curve sections, if they already exist in the hash table the remaining computation can be skipped with the precomputed result returned instead.

   In most cases a check is performed at least twice. Given two candidate solutions for two distinct agents B_1 and B_2, when evaluating the fitness of B_1 it is checked for intersections with B_2 and equally when evaluating the fitness of B_2 B_2 is checked against B_1, thus buy storing the first result, the second check can be skipped.

   In reality, many curve segments are found repeatedly inside of populations and as such many more duplicate checks can be skipped through this method.

4. I implemented a recursion depth limit so as to introduce an upper bound in computation before the function returns false. Without this I found that often the checks would repeat until it was searching for intersections between infinitesimally small curve sections.

I was also able to reduce the runtime of this function via the use of threading. Upon a subdivision two more threaded tasks are spawned and the disjunction of their results returned. The optimal case for this approach is that the intersection occurs early on in the curve as the task results must be fetched sequentially in the order that they were spawned.

> remove this sentence?

4.3 Language Choice

> decide whether to remove this, I am thinking yes but last para is interesting, maybe move to evaluation

I have chosen to implement my approach using the Julia language project[17].

Julia is a relatively new language first developed in 2012 by Jeff Bezanson, Stefan Karpinski and Viral Shah. It is a multi-paradigm language allowing for functional, object oriented and meta programming approaches to problems. I will mainly be using it for it's functional and OO capabilities.

Julia operates using multiple dispatch similar to languages such as Haskell. It interoperates with C and Fortran codebases without the need of middleman bloat. This fact allows it to utilise the extensive high performance C libraries for floating point operations. Julia is eagerly evaluated, uses a Just in time compiler and has a garbage collector.

Julia features a syntax similar to both Python and Matlab with performance on par with C. As I am already very familiar with python and have studied functional programming in a number of modules; I found this language very quick and intuitive to learn and the resulting code to be clean, idiomatic and fast.

A real-world deployment of a system based on my research would undoubtedly be required to run on small, relatively low performance, embedded systems and as such Julia may not be appropriate here. A language such as C or Rust may be used instead.

Julia also has distributed computation facilities. This sort of functionality could be extremely useful in a system such as mine as it could allow for computation to be spread across the agents themselves, removing the need for a central planning center which could be a single point of failure.

4.4 Results

Due to the high dimensionality of my search space, I cannot visualise the objective function of my GA.

My results were achieved by

5 Evaluation

5.0.1 Bézier Curves

> expand on this section, talk about issues of finding intersection, possible GPU applications

Bézier curves have been utlised in this project to encode and represent the route of a vehicle. As mentioned in Section 2.2, there are many reasons I originally selected them for this task. However, over the course of implementation and testing, a number of downsides have been presented.

5.0.2 Cooperative Planning

My solution to the problem of planning n non-colliding routes for n agents was so wrap my existing `GA` function in a cooperative *layer*. This cooperative layer relied on a function for detecting collisions which had extremely high overhead, at one point causing around 50x slowdown in the running time of the function. Detecting intersections between two Bézier curves is a non-trivial task with the best methods taking the same approach of recursive subdivision that I utilised.

> make mention of possible GPU implementations such at in [3], modern enterprise GPUs have around 4000-10000 cores, approximately the max number of curve splits and comparisons in a binary check of depth 6. Being able to do this in a single cycle would result in a huge speedup

6   Conclusion

Bibliography

[1] Rahul Kala. *On-Road Intelligent Vehicles: Motion Planning for Intelligent Transportation Systems / Rahul Kala.* Butterworth-Heinemann is an imprint of Elsevier, Kidlington, Oxford, UK, 2016.

[2] Zixing Cai and Zhihong Peng. Cooperative Coevolutionary Adaptive Genetic Algorithm in Path Planning of Cooperative Multi-Mobile Robot Systems. *Journal of Intelligent and Robotic Systems*, 33(1):61–71, January 2002.

[3] V. Roberge, M. Tarbouchi, and G. Labonté. Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU. *IEEE Transactions on Aerospace and Electronic Systems*, 54(5):2105–2117, October 2018.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, Cambridge, UNITED STATES, 1992.

[5] SN Bernstein. On the best approximation of continuous functions by polynomials of a given degree. *Comm. Soc. Math. Kharkow, Ser*, 2(13):49–194, 1912.

[6] Jens Gravesen. Adaptive subdivision and the length and energy of Bézier curves. *Computational Geometry*, 8(1):13–31, June 1997.

[7] G. Thippa Reddy, M. Praveen Kumar Reddy, Kuruva Lakshmanna, Dharmendra Singh Rajput, Rajesh Kaluri, and Gautam Srivastava. Hybrid genetic algorithm and a fuzzy logic classifier for heart disease diagnosis. *Evolutionary Intelligence*, 13(2):185–196, June 2020.

[8] Mahdi Shariati, Mohammad Saeed Mafipour, Peyman Mehrabi, Masoud Ahmadi, Karzan Wakil, Nguyen Thoi Trung, and Ali Toghroli. Prediction of concrete strength in presence of furnace slag and fly ash using Hybrid ANN-GA (Artificial Neural Network-Genetic Algorithm). *Smart Structures and Systems*, 25(2):183–195, 2020.

[9] Rahul Kala. Optimization-Based Planning. In *On-Road Intelligent Vehicles*, pages 109–150. Elsevier, 2016.

[10] Luis Cruz-Piris, Miguel A. Lopez-Carmona, and Ivan Marsa-Maestre. Automated Optimization of Intersections Using a Genetic Algorithm. *IEEE Access*, 7:15452–15468, 2019.

[11] Stephen A. Vavasis. Quadratic programming is in NP. *Information Processing Letters*, 36(2):73–77, 1990.

[12] Ardalan Vahidi and Antonio Sciarretta. Energy saving potentials of connected and automated vehicles. *Transportation Research Part C: Emerging Technologies*, 95:822–843, October 2018.

[13] Austin Brown, Jeffrey Gonder, and Brittany Repac. An Analysis of Possible Energy Impacts of Automated Vehicles. In Gereon Meyer and Sven Beiker, editors, *Road Vehicle Automation*, Lecture Notes in Mobility, pages 137–153. Springer International Publishing, Cham, 2014.

[14] Zia Wadud, Don MacKenzie, and Paul Leiby. Help or hindrance? The travel, energy and carbon impacts of highly automated vehicles. *Transportation Research Part A: Policy and Practice*, 86:1–18, April 2016.

[15] Chee Yap. Complete subdivision algorithms, I: Intersection of Bezier curves. In *Proceedings of the Annual Symposium on Computational Geometry*, volume 2006, pages 217–226, January 2006.

[16] Danny Hermes. Helper for bézier curves, triangles, and higher order objects. *The Journal of Open Source Software*, 2(16):267, August 2017.

[17] The Julia Programming Language. https://julialang.org/.

Appendices